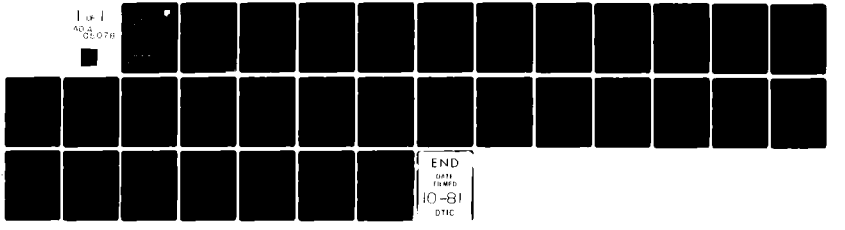
AD-A105 078 POLYTECHNIC INST OF NEW YORK BROOKLYN F/G 9/2
SOFTWARE MODELING STUDIES. VOLUME IV. A STATISTICAL THEORY OF C--ETC(U)
JUL 81 A E LAEMMEL F30602-78-C-0057
UNCLASSIFIED POLY-EE-80-004 RADC-TR-81-183-VOL-4 NL



RADC-TR-81-183, Vol IV (of four)
Final Technical Report
July 1981

SOFTWARE MODELING STUDIES, Volume
A STATISTICAL THEORY OF COMPUTER PROGRAM TESTING

Polytechnic Institute of New York

Arthur E. Laemmel

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-183, Vol IV (of four) has been reviewed and is approved for publication.

APPROVED:

ROCCO F. IUORNO
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

408704

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-81-183, Vol IV (of four)	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): SOFTWARE MODELING STUDIES . V... A STATISTICAL THEORY OF COMPUTER PROGRAM TESTING	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report January 78 — October 80
	6. PERFORMING ORG. REPORT NUMBER: EE-80-004
7. AUTHOR(s): Arthur E. Laemmel	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0057
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Polytechnic Institute of New York, 333 Jay Street, Brooklyn NY 11201	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F, 2304J401
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIE), Griffiss AFB NY 13441	12. REPORT DATE: July 1981; 13. NUMBER OF PAGES: 36
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS (of this report): UNCLASSIFIED; 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program Testing
Testing Strategy
Probability of Failure
Program Correctness
Program Errors
Random Testing
Software Error Models

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Program testing is studied for maximizing the reliability of computer programs. Several formulas are derived to estimate the number of tests required to say that a program being tested is correct with a given probability. Strategies to select those test cases which minimize the probability of program error while keeping the number of tests fixed, are stated.

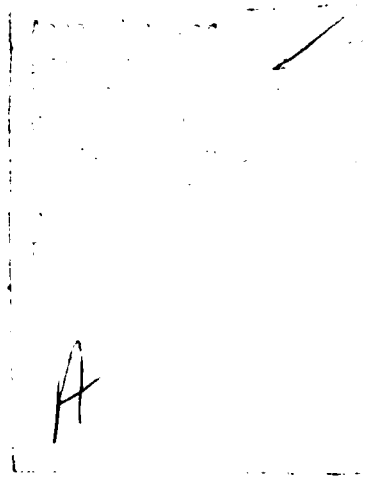
DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408704

TABLE OF CONTENTS



LIST OF FIGURES

A STATISTICAL THEORY OF COMPUTER PROGRAM TESTING

Arthur E. Laemmel

Abstract

Most computer programs are tested with some of the possible sets of input data, but few can be tested with *all* possible input data. Passing such a partial test cannot insure that the program will always function correctly; we can perhaps say that the probability of failure is less than a specified amount. It is the purpose of this report to derive several formulas for the above probability. Also, in some cases an optimum testing strategy can be derived which minimizes that probability of failure.

1.0 Introduction

There are three methods for maximizing the reliability of computer programs: (1) use a systematic procedure which makes it difficult for errors to occur during the writing of the program, (2) prove that the program works correctly by some formal or automatic process, and (3) test and debug the program thoroughly before passing it on to the user. Most programmers will use some combination of these methods, and in fact some procedures involve elements of more than one method. The present report emphasizes testing, but first some remarks will be made about program writing and proving.

(1) Writing Correct Programs: While no one intentionally inserts errors in his program, it is undoubtedly true that many people would produce more reliable programs with less effort if they were taught better programming techniques. However, it seems obvious that the average programmer should test his programs even if he exercises the maximum of care and uses the best techniques.

(2) Proving Program Correctness: There are several reasons why a formal procedure for proving program correctness cannot be relied on to insure absence of errors in practical situations: (i) a uniform algorithm for proving the correctness of an arbitrary program can be shown to be impossible, being essentially equivalent to Turing's halting problem; (ii) even for solvable sub-classes of the correctness-proving problem, the usual method (some improvement on Herbrand search) is so time consuming as to be impractical; (iii) there is always a possibility of error in the proving program, or in applying it to the program being tested.

(3) Testing Computer Programs: In view of the difficulties of validating a computer program by programming techniques or formal proof methods, it is believed that some amount of testing will always be necessary. The purpose of this report is to describe a model which shows the relationship between errors of different types and the probability that they will cause a program to fail, and also to suggest optimum testing methods which minimize the probability of program failure.

Throughout this report it is assumed that a computer program can be tested and that it will either pass or fail the test. It is also assumed that a tester and a user will interpret failure of the program in exactly the same way. For example, failure might mean one of the following:

i) the program "bombs" completely.
ii) an obviously wrong answer is given.
iii) a number is inaccurate in the least significant digit.

iv) the numbers are correct, but the format is wrong.
v) the program works perfectly, but a side effect causes failure in another program.

It must also be decided whether the program alone is being tested, or whether the program and algorithm is being tested. This report is directed mainly to the latter, but the results can be suitably interpreted so as to apply to the former.

The extent to which a computer program (possibly including the underlying algorithm) can be tested varies from application to application and is usually not a yes - no situation. Whether or not a statistical model applies to a particular case depends very strongly on the tester's knowledge of just what answer should be produced by the program. Some of the many possibilities of the user's prior knowledge of the correct answer are:

1. The exact answer is known beforehand. An example would be a math package for a new computer. Accurate tables of cos, arctan, log, etc. have been available for many years.

2. A proposed answer can be checked to see if it is a correct answer, but the correct answer is not known beforehand. Programs which calculate the roots of transcendental equations are examples of this category.

3. Answers for certain special input values are known beforehand. This is very common. For example, if a program is supposed to output the capacitance of an arbitrary two conductor transmission line, it would be natural to test it for coaxial cylinders.

4. The answer is not known beforehand, nor can it be accurately checked for any combination of input values. However, certain consistency relations among different outputs are known. For example, it may be obvious that the program should generate an output which is a monotonically increasing function of the input. A test which detects a decrease indicates an incorrect program, but no amount of testing can indicate a correct program.

5. Absolutely nothing is known about the answers which should be produced. This is probably very rare. Even here, the theory presented in this report applies if the program "bombs," i.e., a fatal or non-fatal run-time error message is produced.

The identification of an error is often not unique. This is illustrated by the fact that error messages from a compiler or run-time monitor can direct the user away from what he considers as the error. For example, if the compiler says "line 15, operator missing" the error might really be "line

14, statement terminator missing." In many cases the syntax can be corrected in several places to get the program thru the compile stage, but one place usually contains the reported error, and another place is usually where the error should be corrected. Similar comments can be made about semantic errors detected at run time. In some cases an error might equally well be corrected in one of several places. For example, a common PL/1 error might be corrected by either declaring K to be a FLOAT number, by using XK instead, or by forcing a necessary type conversion. This type of ambiguity in defining the error is not believed to cause any difficulties with the formulas developed in this report, provided the parameters are interpreted correctly.

2.0 Definitions and Objectives

2.1 Definitions Some basic aspects of the testing process apply equally well to a computer program or to a physical device. For this reason the program or device being tested will sometimes be called the module. After the module has been constructed, it is checked by a tester and then employed by a user. The probability of error, P_e, which occurs during use is given by

$$P_e = P_m P_u \qquad (1)$$

where P_m is the probability that the tester misses all of the residual bugs in the module, and P_u is the probability that the user then encounters one of the overlooked bugs. If exhaustive testing is possible then $P_m = 0$, since it is assumed that if a bug is found it then is corrected and the whole process is started again. In most cases of interest exhaustive testing is not possible or practical. If there are no bugs then all of the probabilities are zero and this possibliity appears as a special case in the analysis to follow. Some alternate definitions of "probability of error" are given in Section

2.2 Objectives Before presenting the detailed mathematical results, the objectives of this work will be stated more explicitly. Roughly, the aims are to provide an estimate of the number of tests required to say that the program being tested is correct with a given probability, and to provide a strategy so that a given number of tests are chosen most effectively.

The first objective is met by deriving a functional relationship between the number of tests and the probability of error. There are several possibilities for defining probability of errors, the one used here might be called the "probability of embarrassment" for the tester. The tester is not embarrassed if

i) he discovers a bug and returns the program to the writer
ii) he approves the program in spite of its having one or more bugs, but the user does not encounter one of the remaining bugs. The tester is embarrassed if
iii) he approves the program and then the user encounters an undetected bug. If the model is designed appropriately, the probability of error decreases as the number of tests is increased according to Eq. 30 below.

The second objective is met by choosing those tests which minimize the expression derived for probability of error (defined as above) while keeping the number of tests fixed. Specifically, the principal question which must be answered here is: should testing be concentrated on input data combinations which are most likely to be chosen by the user, or input data combinations with a large a priori probability of causing program failure? The answer is that tests should be applied to both of these combinations of input data in a ratio which can be calculated from Eq. 33 below.

Actually, several simpler models are also analyzed before that described by Eqs. 30 and 33. All of these expressions contain many parameters, and curves are plotted for selected values. No real data were used for the parameters, but this should certainly be done in the future.

3.0 Models

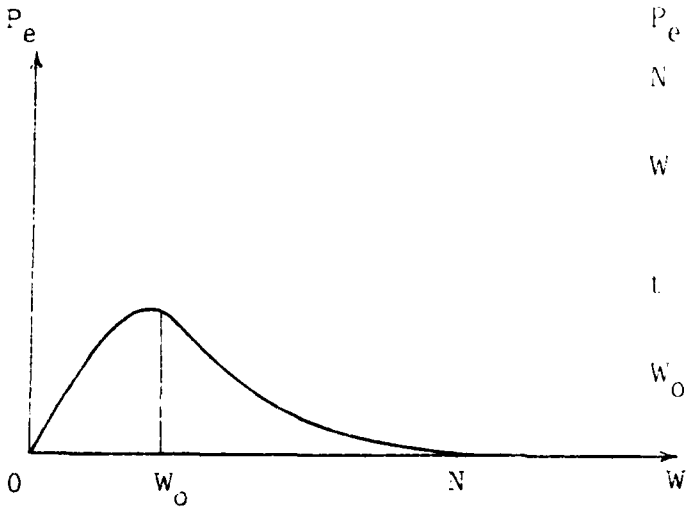
3.1 Elementary Model A simple case might be the following: The module has N possible input values, and each of these are equally likely to be chosen by the tester or by the user. Of these input values, W cause improper functioning of the module, but neither the tester nor the user knows which inputs cause errors or even how large W is. The tester chooses t inputs at random without keeping a record of inputs previously tested, i.e., sampling with replacement. Under these circumstances $P_u = W/N$, $P_m = (1 - W/N)^t$ and

$$P_e = \frac{W}{N}\left(1 - \frac{W}{N}\right)^t \leq \frac{W}{N} e^{-Wt/N} \qquad (2)$$

A plot of P_e vs. W is displayed in Fig. 1. If the testing is to do any good, i.e., to reduce P_e significantly less than W/N, then it is necessary that

$$t >> \frac{N}{W} \qquad (3)$$

In many applications it is found that satisfying the inequality of Eq. (3) requires a very large number of tests t, and that our intuitive feeling is that P_e is acceptably small in spite of testing using far fewer tests than indicated.



P_e = ERROR PROBABILITY

N = NUMBER OF INPUT VALUES

W = NUMBER OF INPUT VALUES CAUSING MALFUNCTIONS

t = NUMBER OF INPUTS TESTED

W_o = WORST VALUE OF W

FIGURE 1. PLOT OF P_e VS. W FROM THE MODEL OF EQUATION 2.

It will continue to be assumed that the tester passes no information to the user about which inputs were tested. It will also be assumed that each test either succeeds or fails and that there is no additional information which would permit sequential sampling methods. *Sampling without replacement* would slightly lower P_m to

$$P_m = \begin{cases} \dfrac{(N-W)!(N-t)!}{(N-W-t)!\,N!} & t < N-W \\ 0 & t > N-W \end{cases} \qquad (4)$$

Note that this method requires that the tester keep a record of inputs already tested, or that he avoids duplication by other means.

The particular type of error to which P_e pertains must be borne in mind to avoid confusion with other possible definitions of error. If W=N all inputs to the module cause malfunctions but $P_e = 0$ according to Equations (2) and (4). This is true because the tester removes user bugs with each test, and continues until all tests are exhausted. In this case the tester always rejects the module (provided only that $t > 0$) and so the user cannot experience a malfunction. Note that P_e is neither the probability {user has a malfunction} nor the probability {user has a malfunction | tester accepts module}. Rather, P_e is the probability {user has a malfunction and tester accepts module}. If the number of tests is fixed at t, and if testing without replacement is done, then the value of W which maximizes P_e is

$$W_o = \frac{N}{t+1} \tag{5}$$

From Eq. (5) or Fig. 1, it can be seen that as $W \to 0$ then $P_e \to 0$ also. This is so because for small W there is a small chance that the user will encounter a faulty input. Similarly as $W \to N$ then $P_e \to 0$ also because there is little chance that the tester will accept the module. Of course, the last case is undesirable for reasons other than the value of P_e, i.e., the user has a small probability of receiving a released program.

3.2. Model with Unequal Probabilities The model described in the preceding section is too simple to apply to most practical testing situations: some inputs are more likely to fail than others, and the probability of one input failing may not be independent of another input failing. Often a single bug may cause many inputs to fail. The tester may not choose the inputs to be tested randomly, but rather in such a way as to utilize his knowledge of the prior failure probabilities. The user may not be free to choose more reliable inputs; in fact, he may be constrained by the problem to use less reliable inputs. The model to be described here includes three events, the last two being independent of each other but dependent on the first.

(1) A programmer constructs a module which has an error pattern α with probability $P(\alpha)$. α might be a binary vector $(\alpha_1, \alpha_2, \ldots, \alpha_N)$ with $\alpha_i = 1$ meaning input i fails and $\alpha_i = 0$ meaning input i functions correctly.

(2) A tester tries certain inputs to the module and accepts the module with probability R(accept | α). The tester passes no information concerning which inputs were tested to the user.

(3) A user selects one of the inputs and the module fails with probability Q(fail | α). The error probability defined previously is now given by

$$P_e = \sum_{\alpha \in A} P(\alpha)\, Q(\text{fail} \mid \alpha)\, R(\text{accept} \mid \alpha) \tag{6}$$

where A is the set of all possible error patterns.

To illustrate, if there are N input values then A consists of 2^N elements. Assume $P(\alpha)$ is 0 for all A except for the first w:

$$\alpha_w = (\underbrace{1,1,\ldots,1}_{w}, \underbrace{0,0,\ldots,0}_{N-w}) \tag{6a}$$

and let the probability of the user selecting input i be q_i. Then

$$\underline{Q}(\text{fail} \mid \alpha_W) = \sum_{i=1}^{W} q_i$$

Assume the tester selects his inputs randomly, choosing input i with probability r_i.* Then

$$\underline{R}(\text{accept} \mid \alpha_W) = \prod_{j=1}^{W} (1 - r_j) \tag{7}$$

$$P_e = \left(\sum_{i=1}^{W} q_i \right) \prod_{j=1}^{W} (1 - r_j)$$

If $q_i = 1/N$ and $r_j = t/N$ this reduces (approximately) to the elementary model given above:

$$P_e = \frac{W}{N} \left(1 - \frac{t}{N}\right)^W \tag{8}$$

Another, more useful, form for $P(\alpha)$ than Eq. (6a) is obtained by assuming that the i^{th} input malfunctions with probability p_i. Then

$$\underline{P}(\alpha) = \prod_{i=1}^{N} p_i^{\alpha_i} (1 - p_i)^{1-\alpha_i}$$

$$\underline{Q}(\text{fail} \mid \alpha) = \sum_{j=1}^{N} \alpha_j q_j$$

$$\underline{R}(a \mid \alpha) = \prod_{i=1}^{N} (1 - r_j)^{\alpha_i}$$

---

*The formula also applies if r_i is given the interpretation "input i is tested and the response is noted to be wrong by the tester." Specifically $r_i = 0$ might mean either that input i was not tested, or that it was tested and an error was not detected.

The two products can be combined in evaluating P_e:

$$P_e = \sum_{\alpha_1} \sum_{\alpha_2} \cdots \sum_{\alpha_N} \sum_{j=1}^{N} \alpha_j q_j \prod_{i=1}^{N} F_i(\alpha_i)$$

where

$$F_i(\alpha_i) = p_i^{\alpha_i} (1-p_i)^{1-\alpha_i} (1-r_i)^{\alpha_i}$$

This reduces to

$$P_e = \prod_{j=1}^{N} (1-p_j r_j) \sum_{i=1}^{N} q_i \frac{p_i(1-r_i)}{1-p_i r_i} \qquad (9)$$

If the tester selects t inputs deterministically, and if the inputs are permuted so that these occur first, then

$$P_e = \prod_{j=1}^{t} (1-p_j) \sum_{i=t+1}^{N} p_i q_i \qquad (10)$$

An optimum testing strategy is obtained if the inputs are permuted so that the expression is minimized. The first factor suggests testing inputs with the largest p_i, but the second factor suggests testing inputs with the largest $p_i q_i$. Let the above expression be abbreviated as $P_e = P_m P_u$ and note that P_u is the probability of the user getting an error on an untested input. Consider the effect of adding one more input to the test.

Let

$$P'_e = P_m(1-p_k)(P_u - p_k q_k)$$

$$P'_e \simeq P_e \left(1 - \frac{(q_k + P_u)p_k}{P_u}\right)$$

Thus, the criterion is to select the input with the largest

$$(q_k + P_u)p_k \qquad (11)$$

As can be seen, this provides a weighted compromise between selection on the basis of p_k or $q_k p_k$ alone.

3.3 Model with Statistical Dependence Computer programs usually fail for a whole set of input values as a result of a single bug. It is more

realistic to assume that failures due to different bugs are statistically independent rather than failures for different input values. For example, a single oversight might cause a square root program to fail for all negative numbers. Let

$$\sigma_{ij} = \begin{cases} 1 \text{ if the } j^{th} \text{ bug causes failure for input } i \\ 0 \text{ if the } j^{th} \text{ bug doesn't affect input } i \end{cases}$$

If β_j is the probability of the j^{th} bug, if M is the number of possible bugs, and if θ_j $(j = 1,2,\ldots,M)$ is the pattern of actual bugs, then (note definitions of $\underline{P}$, $\underline{Q}$, $\underline{R}$ in the probability of event statements (1), (2), and (3) of Section 3.2)

$$P_e = \sum_{\theta} \underline{P}(\theta)\, \underline{Q}(f \mid \theta)\, \underline{R}(\alpha \mid \theta) \qquad (12)$$

This is analogous to the corresponding formula in a given above. Here,

$$\underline{P}(\theta) = \prod_{i=1}^{M} \beta_i^{\theta_i} (1-\beta_i)^{1-\theta_i}$$

$$\underline{Q}(f \mid \theta) = \sum_{j=1}^{N} q_j \left[1 - \prod_{k=1}^{M} (1-\sigma_{jk}\theta_k)\right] \qquad (13)$$

$$\underline{R}(\alpha \mid \theta) = \prod_{i=1}^{M} \prod_{\ell=1}^{N} (1-r_\ell)^{\sigma_{\ell i}\theta_i}$$

Combining and rearranging gives

$$P_e = \sum_{j=1}^{N} q_j \sum_{\theta} A_j(\theta_1, \theta_2, \ldots, \theta_M) \prod_{i=1}^{M} \Gamma_i(\theta_i)$$

where

$$\Gamma_i(\theta_i) = \beta_i^{\theta_i} (1-\beta_i)^{1-\theta_i} \prod_{\ell=1}^{N} (1-r_\ell)^{\sigma_{\ell i}\theta_i}$$

and

$$A_j(\theta_1, \ldots, \theta_M) = 1 - \prod_{k=1}^{M} (1-\sigma_{jk}\theta_k)$$

The summations over θ_i are for only two values (0,1) and can be carried out to give

$$P_e = \prod_{i=1}^{M} [1-\beta_i+\beta_i \prod_{\ell=1}^{N} (1-r_\ell)^{\sigma_{\ell i}}] - \sum_{j=1}^{N} q_j \prod_{i=1}^{M} [1-\beta_i+(1-\sigma_{ji})\beta_i \prod_{\ell=1}^{N} (1-r_\ell)^{\sigma_{\ell i}}] \quad (14)$$

If deterministic testing is used (r=0,1) over the first t inputs:

$$P_e = \prod_{i=1}^{M}{}' (1-\beta_i) \sum_{j=t+1}^{N} q_j [1 - \prod_{i=1}^{M}{}'' (1-\sigma_{ji}\beta_i)]$$

Where Π' refers to a product over terms involving a value of i such that $\sigma_{ij} = 1$ for at least one value of j in the range j = 1,2,...,t, and where Π'' refers to the other values of i. The bugs can be permuted so that $1 \leq i \leq \tau$ implies $\sigma_{ji} = 1$ for at least one value of j from 1 to t and $\tau < i \leq M$ implies $\sigma_{ji} = 0$ for all values of j from 1 to t. Then

$$\prod_{i=1}^{M}{}' = \prod_{i=1}^{\tau} \quad \text{and} \quad \prod_{i=1}^{M}{}'' = \prod_{i=\tau+1}^{M}$$

The problem is then to minimize

$$P_e = \prod_{i=1}^{\tau} (1-\beta_i) \sum_{j=t+1}^{N} q_j [1 - \prod_{i=\tau+1}^{M} (1-\sigma_{ji}\beta_i) \quad (15)$$

A graphical interpretation of the above is portrayed in Fig. 2, and illustrates how different inputs excite various bugs. For example, input 2 excites bugs numbered 6,7,8 and 9. Bug numbered 8 can only be discovered through the application of inputs 2 or 4.

A clearer picture of how P_e depends on the amount of testing, t, can be obtained by deriving an upper bound from Eq. (15). For most cases of interest this upper bound will be tight, i.e., a good approximation. Multiply thru by the first product in Eq. (15):

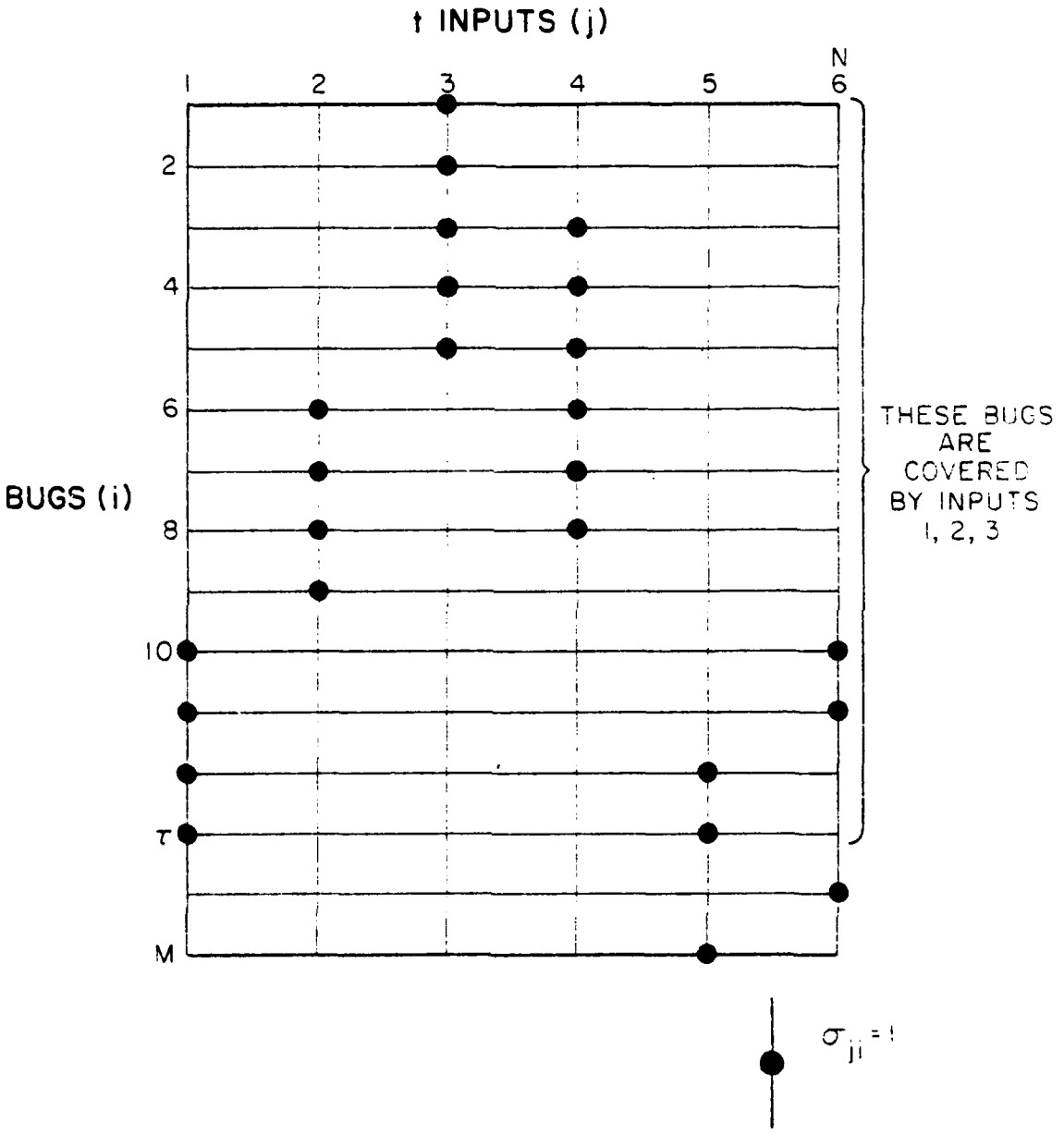


FIGURE 2. PLOT OF σ_{ji}

$$P_e = \sum_{j=t+1}^{N} q_j \left[\prod_{i=1}^{\tau} (1-\beta_i) - \prod_{i=1}^{M} (1-\gamma_{ji}\beta_i) \right] \qquad (16)$$

where

$$\gamma_{ji} = \begin{cases} 1 & 1 \le i \le \tau \\ \sigma_{ji} & \tau < i \le M \end{cases}$$

Now observe the following:

$$- \prod_{i=1}^{M} (1-\gamma_{ji}\beta_i) \quad - \prod_{i=1}^{M} (1-\beta_i) \quad \text{since} \quad \gamma_{ji} \le 1$$

$$\sum_{j=t+1}^{N} q_j - Q(t) \le 1 \quad (\text{note } Q(t) = 1 - \frac{t}{N} \approx 1 \text{ if } q_j = \frac{1}{N})$$

$$\prod_{i=1}^{\tau} (1-\beta_i) \le e^{-\sum_{i=1}^{\tau} \beta_i}$$

Combining these:

$$P_e \le \bar{P}_e = e^{-\sum_{i=1}^{\tau} \beta_i} - R_o \qquad (17)$$

where

$$R_o = \prod_{i=1}^{M} (1-\beta_i) \approx e^{-\sum_{i=1}^{M} \beta_i}$$

From the way the σ_{ji} matrix was permuted, it can be seen that $t \to M$ (monotonically), and from Eq. (16) these, in turn, imply $P_e \to 0$ (monotonically).

If Eq. (17) is a good approximation to Eq. (16), then the rate at which $P_e \to 0$ can be seen to be exponential.

3.4 Illustrative Special Case A simple special case will illustrate the above formulas, and is useful in getting a rough idea of the number of tests and the probability of error. Suppose that each of the M bugs occurs with the same probability of error, β, and affects the same number of input values, b. Suppose further, that the pattern of input errors is the most difficult to detect with the given number of tests, t, i.e., that the error subsets are as "disjoint" as possible. If the tests are distributed most effectively over the inputs, each test will detect Mb/N bugs. The testing will result in M - Mbt/N undetected bugs, and τ = Mbt/N (assuming bt $\leq$ N). Thus

$$P_e \approx e^{-\frac{\beta Mbt}{N}} = e^{-\beta M\tau} \qquad (18)$$

As $\tau \to$ N/b this shows $P_e \to 0$. In order to make P_e small compared to its value with no testing, $1 - e^{-\beta M\tau}$ must be made comparable to 1. For more general assumptions concerning the parameters, and with a good choice of tests, t might be a much smaller fraction of N.

3.5 Optimum Testing Strategy (in Section 3.3) For a given t, that selection of inputs to be tested which minimizes P_e will be defined as the optimum testing strategy. From Eq. (15), and intuition, one might be tempted to test the t inputs most likely to be chosen by the user, thus minimizing the factor Σq. However, for most cases of interest, this approach would be futile. If the q_j are even very roughly equal, the number of possible inputs N is so huge that it would be impossible to test enough of them to reduce

$$\sum_{t+1}^{N} q_j$$

significantly below unity. Essentially, the q_j only influence the testing strategy thru the fact that the square bracketted term in Eq. (15) depends on i.

The first term in Eq. (15) is the most important for values of the parameters of most interest, and as was shown above, it is minimized (approximately) by choosing test values to maximize

$$E = \sum_{i=1}^{\tau} \beta_i \tag{19a}$$

$$E = \tau\beta \qquad \text{if} \qquad \beta_i = \beta \tag{19b}$$

If the β_i are equal, P_e is then minimized (approximately) by maximizing τ. This is essentially the covering problem of switching theory and operations research.[1,2] A set of inputs $i \in I$ is said to cover a set of bugs $j \in J$ if $\sigma_{ji} = 1$ for every j in J for at least one i in I. Referring to Fig. 2, input 4 covers bugs 3, 4, 5, 6, 7 and 8; and input pair 1,4 covers bugs 3, 4, 5, 6, 7, 8, 10, 11, 12 and 13. These inputs are optimum in that they cover the most bugs possible. Thus, using these optimum choices, $\tau = 6$ and 10 for $t = 1$ and 2. However, the optimum set of 3 inputs is not obtained by adding that input which would cover the most additional bugs. Inputs 1, 3, 4 cover only 12 bugs, but inputs 1, 2, 3 cover 13 bugs. The covering problem is usually stated as minimizing the number of tests required to cover all bugs (in the present terminology), and no general algorithm for its solution is known. The process described above, repeatedly adding that test which causes the greatest increment in bugs covered, is usually referred to as the "heaviest-first algorithm."

A sketch of τ vs. t for the present example is shown in Fig. 3.

Note that the number of covered bugs added at each step is a decreasing function of t. If t is very large, the bugs being covered might be those affecting only a single input, such as divide-by-zero bugs.

The rate of increase of P_e with increasing t is certainly of great interest, and one simple functional dependence can be given using the above ideas. Define $\tau_1, \tau_2, \ldots$ as those values corresponding to t=1,2,... when optimum or near-optimum testing strategy is used. Eq. (19a) can be rewritten

$$E_t = \sum_{i=1}^{\tau_t} \beta_i = \sum_{k=1}^{t} B_K \tag{20}$$

where

$$B_K = \sum_{\tau_{k-1}+1}^{\tau_k} \beta_i, \qquad \tau_0 = 0$$

From Eq. (15) with the second and third terms approximated by unity, or from Eq. (17) with R_o approximated by zero:

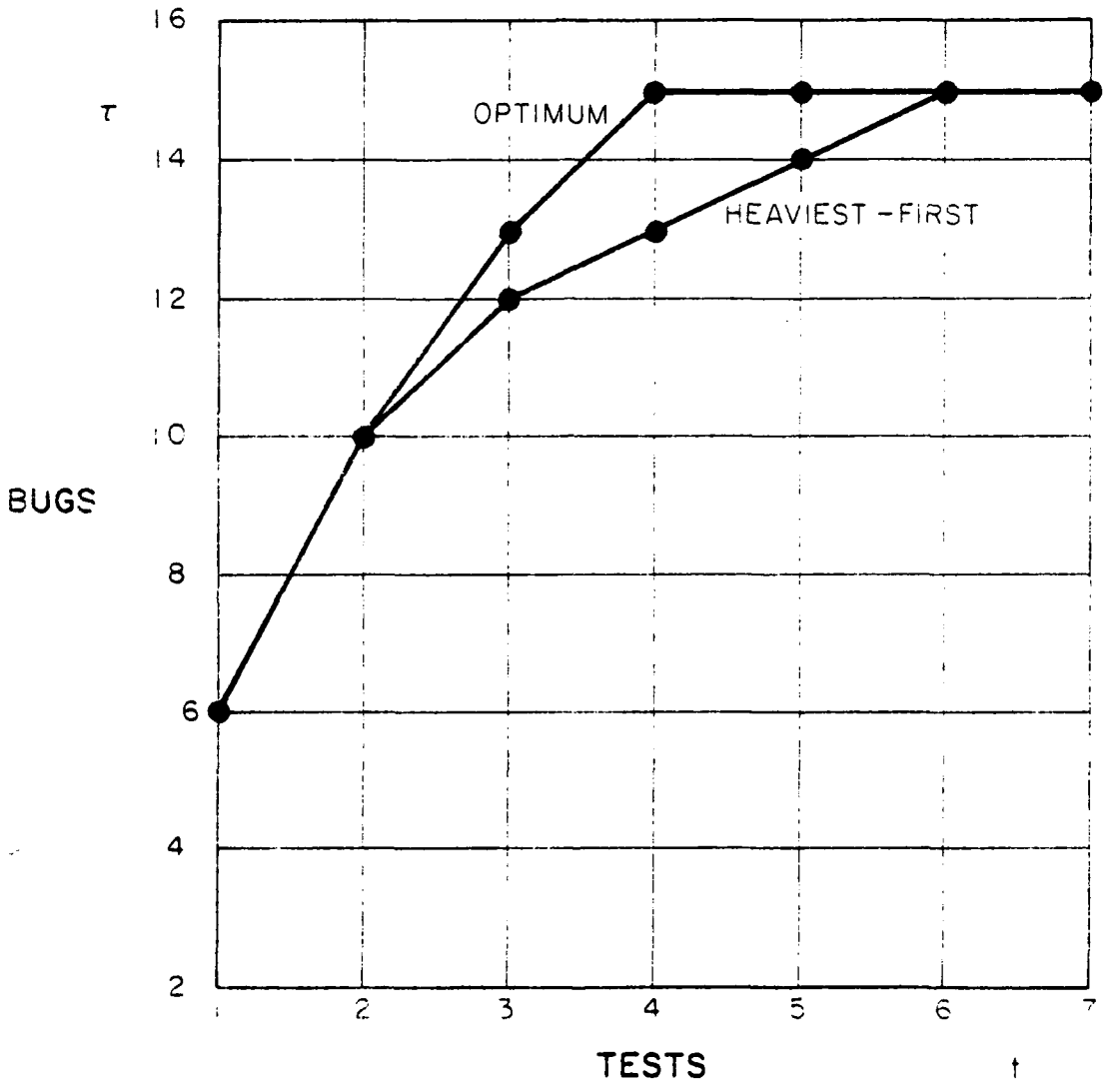


FIGURE 3. BUGS COVERED VS. NUMBER OF TESTS

$$P_e(t) \approx e^{-\sum_{k=1}^{t} B_K} \qquad (21)$$

Now if B_k is assumed to have a form which is a decreasing function of k and which can be finitely summed in closed form, a convenient relation can be found. For example, if $B_k = c/k$

$$\sum_{k=1}^{t} \frac{c}{K} \approx 0.5772c + c\ell nt$$

$$P_e(t) \approx e^{-(0.5772c + c\ell nt)} = t^{-c} \qquad (22)$$

This shows what might be expected in practice, a gradual decrease of P_e over many decades as t is increased over several decades. When some relations are plotted, eg. Eq. (2) or Eq. (18), they exhibit very sudden and deep drops in P_e when certain values of t are approached, and this is not the behavior to be expected except in the simplest programs. Of course, Eq. (22) does not have enough parameters, but similar formulas can be found which do have enough parameters.

3.6 Interpretation of a Program Bug There are many possible ways to interpret a bug.

1. Formally, a bug is simply a subset of input values which fail together when the program is used, e.g., the region $B^2 - 4\,AC < 0$ in solving a quadratic.

2. A bug might be a careless typing error, such as A+B instead of A-B.

3. A bug might be an incorrect statement or a defective subroutine.

4. A more flexible definition of bug should allow such things as omitting a check for dividing by zero. These bugs of omission are harder to handle with fixed M and matrix σ_{ji}.

5. A bug might be defined as a distinct path thru a flowchart. A popular testing procedure is to run at least once thru every such path, and if these are the only bugs included, this constitutes a complete cover and $P_e = 0$. Unfortunately, the model would then not be very realistic because P_e would never actually vanish except in the most trivial programs.

6. If programming can be identified with decisions, a bug is a wrong decision.

4.0 Random Testing

4.1 Random Test Models Some of the difficulties involved in relating the number of bugs discovered and the number of tests made can be avoided if the tests are chosen randomly rather than deterministically. Optimum strategies can still be found if the probabilities of choosing the inputs to be tested are not equal. The probability of error P_e for random testing has already been found in Eq. (14) above. Let s_ℓ be the probability that the tester chooses the ℓ'th input at any particular test. If the number of tests is t, then the probability that the ℓ'th input is tested during the series of t tests is

$$r_\ell = 1 - (1 - s_\ell)^t \tag{23}$$

Note that the s_ℓ sum to unity, but that each r_ℓ can range from zero to one. It is helpful to define

$$A_i = \prod_{\ell=1}^{N} (1-s_\ell)^{\sigma_{\ell i}} \tag{24}$$

which is the approximate probability of a single test missing the i'th bug. Note that A_i is certainly no greater than unity. Eq. (14) can now be rewritten in terms of s, A and t.

$$P_e(t) = \prod_{i=1}^{M} (1-\beta_i+\beta_i A_i^t) \sum_{j=1}^{N} q_j \left[1 - \prod_{i=1}^{M} \left(1 - \frac{\sigma_{ji}\beta_i A_i^t}{1-\beta_i+\beta_i A_i^t}\right)\right] \tag{25}$$

As the number of tests is increased the probability of error approaches zero

$$\lim_{t\to\infty} P_e(t) = 0$$

If no tests are made

$$P_e(0) = 1 - \sum_{j=1}^{N} q_j \prod_{i=1}^{M} (1-\sigma_{ji}\beta_i) \tag{26}$$

It is easier to see the importance of different terms in Eq. (25) by examining the asymptotic form for large t:

$$\bar{P}_e(t) = \prod_{h=1}^{M} (1-\beta_h) \sum_{i=1}^{M} \frac{\beta_i}{1-\beta_i} Q_i A_i^t \qquad (27)$$

where

$$Q_i = \sum_{j=1}^{N} q_j \sigma_{ji}$$

The quantity Q_i is the probability that the user encounters an error from the i'th bug. Since each s_ℓ is very small an exponential can be used to express A:

$$A_i = \prod_{\ell=1}^{N} (1-s_\ell)^{\sigma_{\ell i}} = e^{-S_i} \qquad (28)$$

where

$$S_i = \psi \sum_{\ell=1}^{N} s_\ell \sigma_{\ell i} \qquad 1 \le \psi \le \frac{1}{1 - s_{max}} \qquad (29)$$

Note that if the tester uses the same probabilities as the user then $S_i=\psi Q_i$. If this assumption is made, if the β are small, and if Q_i takes on only a few distinct values $Q_1, Q_2, \ldots, Q_n$, then the asymptotic form of Eq. (17) can be approximated by

$$\bar{P}_e(t) \tilde{\sim} \sum_{k=1}^{n} \beta_k N_k Q_k e^{-Q_k t} \qquad (30)$$

where

N_k is the number of bugs with a probability of discovery Q_k (These bugs which have probility Q_k will be called a "block".)

and

β_k is the average probability of bugs in the k'th block being introduced by the program writer.

No loss of generality is obtained if the Q_k are arranged in order of decreasing magnitude. Eq. (30) clearly represents a decreasing function of t, but it is not so apparent that the decrease is more than 1/t than exponential over the range of interest. To see this, note that each term in Eq. (30) reaches a maximum as a function of Q_k at $Q_k = 1/t$. For each

t a certain term in the sum will be dominant and values of k both larger and smaller than this dominant term will contribute relatively small amounts. This leads to an approximate formula

$$\bar{P}_e(t_k) \simeq \frac{\beta_k N_k}{t_k e} \tag{31}$$

where

$$t_k = \frac{1}{Q_k}$$

If the product $\beta_k N_k$ is approximately independent of k, then P_e will decrease approximately as $1/t_k$. The decrease is exponential below $t=1/Q_{min}$, but this value could be very large indeed. For example, if the input is a 32 bit number, and if a bug causes an error for a single input value, then $Q_{min} = 2^{32}$.

Such a bug would be very hard to detect by random testing, but by the same token it would be very unlikely to cause a user error unless the user favors that value for some reason. The behavior of Eqs. (30) and (31) is sketched in Fig. 4 and Fig. 5.

Fig. 5 shows the way in which the probability of error P_e decreases with the number of tests t. The parameters were chosen to illustrate the following cases:

i) P_e decreases slowly at first, then rapidly

ii) P_e decreases gradually throughout the range

iii) P_e decreases rapidly with the first few tests, then many more tests are required to obtain a further decrease.

As might be expected, in case (i) above most bugs are difficult to detect, in case (ii) bugs are evenly distributed, and in case (iii) bugs are either very easy or very difficult to detect.

Program bugs have been divided into four, and in some cases five, classes, indicated by k in the following

Q_k is the probability of detecting a class k bug

β_k is the probability that a user has introduced a class k bug

N_k is the number of possible class k bugs.

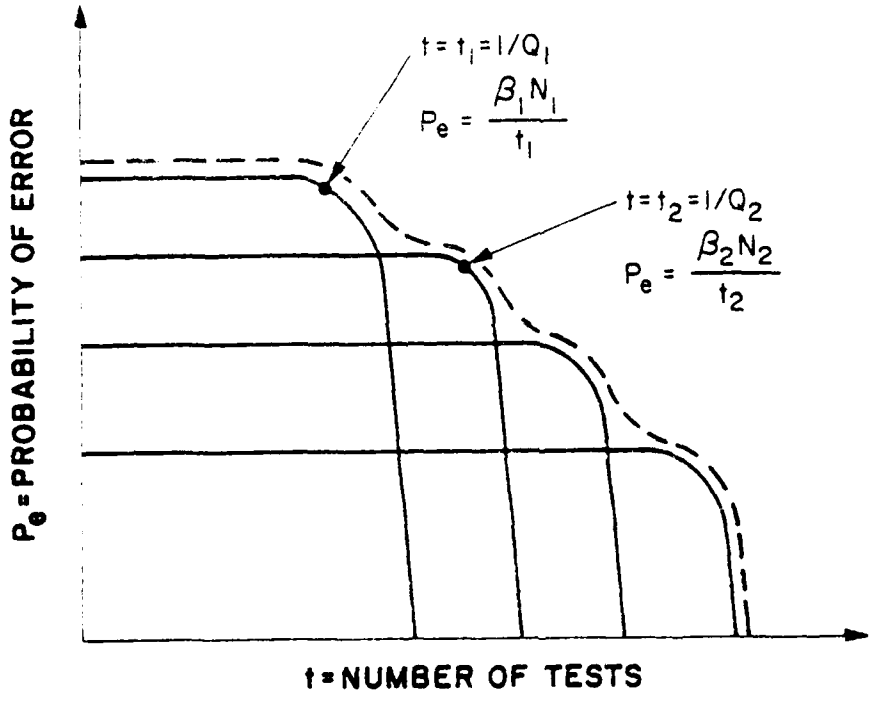


FIGURE 4. PROBABILITY OF ERROR VS. NUMBER OF (RANDOM) TESTS

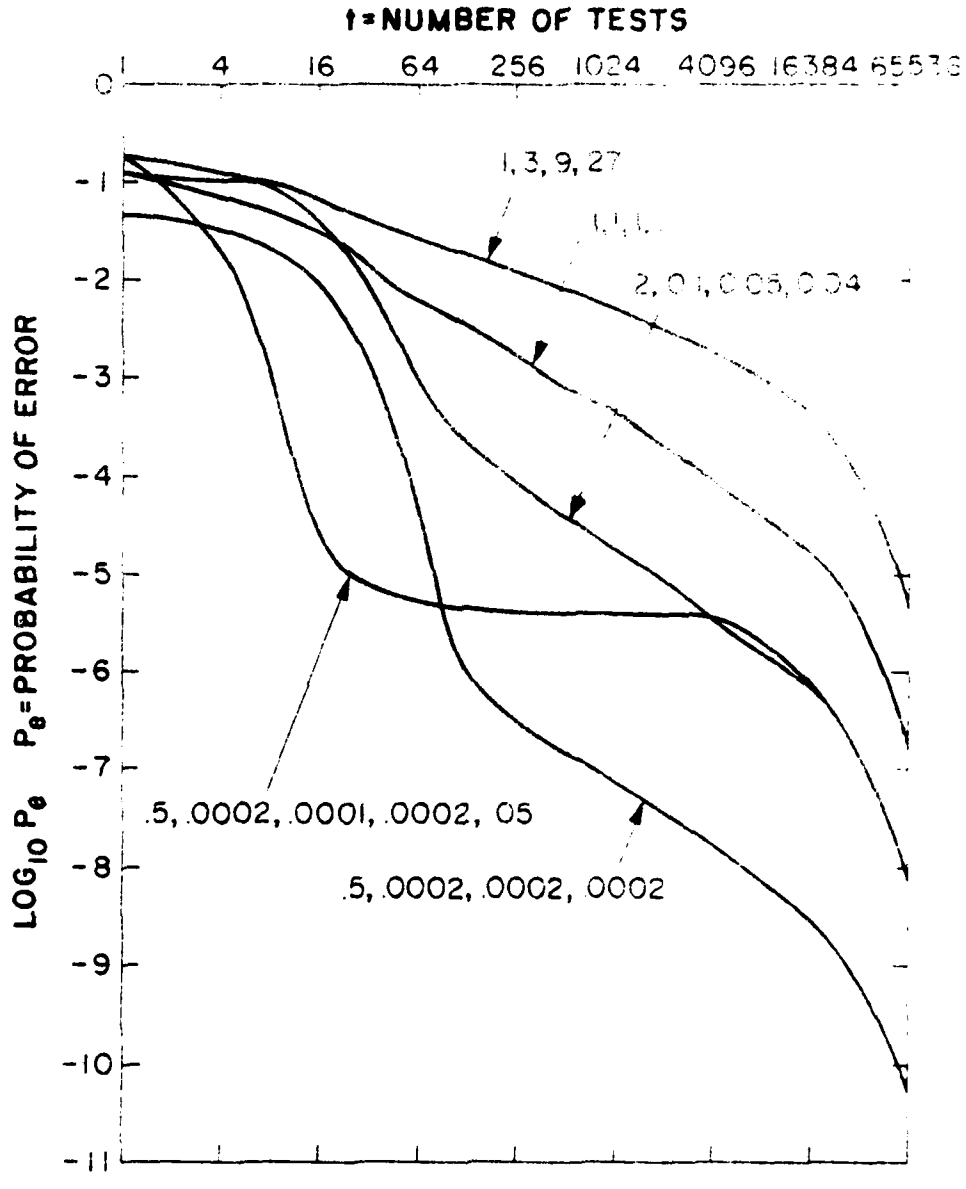


FIGURE 5. SPECIFIC EXAMPLES OF ERROR PROBABILITY CURVES (from Eq. (30), see text for explanation of parameters)

The parameter Q_k is always 0.1, .01, .001 and .0001, except where 5 classes are present in which case Q_k is 0.7, 0.1, .01, .01, .001, and .0001. Shown on the curves is the product $\beta_k N_k$ for each of the four or five classes. For example, the curve marked 1,1,1,1 applies to these 4 cases:

Q_k	β_k	N_k
0.1	.05	20
.01	.05	20
.001	.05	20
.0001	.05	20

Q_k	β_k	N_k
0.1	.01	100
.01	.005	200
.001	.002	500
.0001	.001	100

Q_k	β_k	N_k
0.1	.01	100
.01	.001	1000
.001	.001	1000
.0001	.01	100

Q_k	β_k	N_k
0.1	.002	500
.01	.01	100
.001	.01	100
.0001	.002	500

Ease of detection, as reflected in Q_k, depends largely on the number of inputs a bug effects.

4.2 Optimum Testing Strategy (in Section 4.1) Even though the inputs to be tested are to be selected randomly, an optimum strategy exists in that the probabilities of testing various inputs can be chosen so as to minimize P_e. The asymptotic form expressed by Eq. (27) can be rewritten

$$\bar{P}_e(t) = (1-\beta_n) \sum_{i=1}^{M} \frac{\beta_i}{1-\beta_i} Q_i e^{-ts_i} \qquad (32)$$

The quantities β_i and Q_i are fixed by the problem, but the s_i can be chosen to minimize $\bar{P}_e$, at least insofar as they depend on the s_ℓ (see Eq. (29)).

The optimum s_ℓ can easily be found for the special case where $\sigma_{ji} = \delta_{ji}$, i.e., where each bug affects only one input. The problem is to minimize the asymptotic form (with some additional approximations):

$$f = \sum_{i+1}^{N} \beta_i q_i e^{-s_i t}$$

while keeping constant

$$1 = g = \sum_{i=1}^{N} s_i$$

The method of Lagrange multipliers gives the equation

$$0 = \frac{\partial f}{\partial s_k} + \lambda \frac{\partial g}{\partial s_k} = - \beta_k q_k t e^{- s_k t} + \lambda$$

After eliminating λ to insure that the s_i sum to unity, this gives

$$s_k = \frac{1}{N} + \frac{1}{t} \ell n \ (c \beta_k q_k) \tag{33}$$

where

$$\ell n \ c = - \frac{1}{N} \sum_{i=1}^{N} \ell n \ \beta_i q_i$$

This solution shows that if a small number of tests are made, the tester should favor the inputs most likely to be used and most likely to be in error, but that if a large number of tests are to be made the tester should select the inputs with an essentially uniform distribution. Even where a non-uniform distribution of tests is indicated by Eq. (33), the tester should distribute his tests more uniformly than the user because of the logarithm (assuming the β_k do not deviate too markedly from uniformity).

The optimum testing strategy when σ_{ji} is a delta functon (each bug affecting only one input) can be quite misleading for the more general case. It may be quite impossible to make the s_i in Eq. (32) either independent of i, or (even logarithmically) proportional to the Q_i. Also, if one returns to the more general Eq. (25) or Eq. (17) and attempts to choose the s_ℓ to minimize P_e or $\bar{P}_e$, it is found that the best value for s_ℓ is either zero or one and that these values depend on an unrealistically detailed knowledge of $\sigma_{\ell i}$. This is really the same as the optimum deterministic testing discussed in Section 3.5.

5.0. Alternative Definitions of Error

The definition of error which was used above may not be suitable for all purposes, but related probabilities can easily be calculated from the equations given. Define two events as follows:

E_u is the event that a user employs the module once and encounters a failure.

E_m is the event that a tester operates the module t times and does not encounter a failure, i.e., the tester accepts the module. The subscript m is mnemonic for "missed," since bugs are always assumed to be present with some probability, and therefore $Pr\{E_m\}$ is the probability that the tester has missed all bugs, i.e., incorrectly accepted the module.

Three different quantities which might be interpreted as the probability of user error are tabulated below (Table I) for the various models which have been analyzed. The effectiveness of the testing process can be judged by comparing the first with the last two. Note that the last, $Pr\{E_u|E_m\}$ is always greater than the second, $Pr\{E_u E_m\} = P_e$. This fact might lead to a misuse of conditional probability, $Pr\{E_u|E_m\}$, in some cases where P_e is small due to tester rejection and where there is a large probability of module bugs.

One is tempted to regard the goal here as minimization of $Pr\{E_u\}$, regarding the avoidance of user error as of most importance to the user. But remember this report seeks an optimum testing method, and that this is different from and does not preclude previously using an optimum programming method. The latter will minimize user error, but will not generally remove the need for testing in addition.

6.0 Conclusions and Comments

It is believed that defining what is meant by the probability of a program error, and presenting a model which permits its exact calculation (Eq. (15)) will provide the nucleus around which a theory of software reliability can be built. The purpose is not merely to get a formula into which numbers can be plugged to give a probability of error -- this does nothing to reduce errors. Rather it is anticipated that by classifying different types of bugs, errors and test results, and by showing how they interact programming systems can be improved and optimum testing procedures can be found.

Most reports on program testing select the test data so as to traverse each path in the program at least once.[3,4] The approach described in this report might seem to be directed at selecting test data according to the problem specification. A near-optimum strategy may be to select test data randomly, since this seems to be a fairly efficient solution to the covering problem, but to also make sure some test points are in various distinct data domains (e.g., $B^2 - 4AC < 0$) and to also make sure each program path is traversed. In addition, it is desirable to concentrate test data at points known to be more likely to be selected by the user. These requirements are often contradictory, but optimum compromises are suggested by Eq. (11), Section 3.5, and Section 4.2 above (provided the stated assump-

tions are met and the required parameters are available). Also formulas derived give some idea of how many tests need to be made to achieve a given reliability.

If further work is to be done along the lines of this report it is suggested:

1) Data be gathered so as to verify the derived relations between the number of tests and the error probability, especially as in Section 4.0.

2) Another model can be developed in which a partial knowledge of the bug-input matrix σ is assumed. This would lead both to a more easily applied optimum testing strategy, and to a method of sequential testing in which the last test points are chosen according to the results of the first test.

7.0 References

1) M.L. Balinski, "Integer Programming: Methods, uses, computations," Management Sci., Vol. 12, p. 253, November 1965.

2. Min-Wen Du, "A way to find a lower bound for a minimal solution of the covering problem," IEEE Trans. on Computers, Vol. 21, p. 317, 1972.

3. J.C. Huang, "An approach to program testing," Comput. Surv., Vol. 7, p. 113, September 1975.

4. W. Miller and D.L. Spooner, "Automatic generation of floating-point test data," IEEE Trans. on Software Engr., Vol. SE2, p. 223, September 1976.

Table 1 summarizes the models discussed.

TABLE 1. Summary of the statistical models.

Probability of	user failure	user failure and acceptance	user failure and acceptance
Model	$\Pr\{E_u\}$	$P_e = \Pr\{E_u E_m\}$	$\Pr\{E_u \mid E_m\}$
Equal failure probabilities $P_i = \frac{W}{N}$	$\frac{W}{N}$	$\frac{W}{N}\left(1 - \frac{W}{N}\right)^t$	$\frac{W}{N}$
Unequal failure probabilities $M=N, \sigma_{ji}=\delta_{ji}$	$\sum_{i=1}^{N} q_i p_i$	$\sum_{i=t+1}^{N} q_i p_i \prod_{j=1}^{t} (1-p_j)$	$\sum_{i=t+1}^{\cdot} q_i p_i$ (sum over untested inputs)
With statistical dependence	$\sum_{j=1}^{N} q_j \left[1 - \prod_{i=1}^{M} (1-\sigma_{ji}\beta_i)\right]$	see Eq. (15)	$\sum_{j=t+1}^{N} q_j \left[1 - \prod_{i=t+1}^{M} (1-\sigma_{ji}\beta_i)\right]$
Special case of above	$1 - (1-\beta)^M$	see Eq. (16)	$\left(1 - \frac{t}{N}\right)\left[1-(1-\beta)^{M\left(1-\frac{t}{N}\right)}\right]$
Random testing	$\sum_{j=1}^{N} q_j \left[1 - (1-\sigma_{ji}\beta_i)\right]$	see Eq. (25)	summation part of Eq. (25): $\sum q_i [\ldots]$
Approximation of above		$\sum_{k=1}^{n} p_k \lambda_k \omega_k e^{-\omega_k t}$	

β Types of Error Probability

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.